CMPSC-F265 Midterm Exam 2
Sample Practice

NAME:	Student ID:

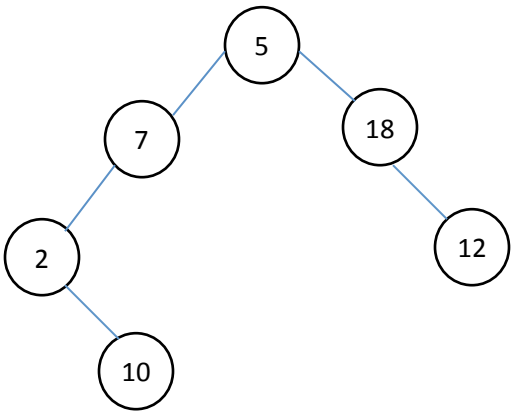
Rules:

- 75 minutes, 100 possible points
- Closed book/notes
- No calculators, no electronic devices

Problem	Score
1	
2	
3	
4	
Total (100)	

Problem 1. Binary Tree

a). Given the following binary tree:



(1) Is this tree a complete binary tree? (Y/N) _____N____

(2) What is the height of the tree? _____3___

(3) What is the depth of node "7" ____1_____

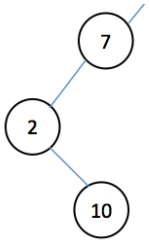
(4) Show the Pre-Order traversal sequence of numbers:
root-left-right:
5 7 2 10 18 12

(5) Show the In-Order traversal sequence of numbers:
left – root – right:
2 10 7 5 18 12

(6) Show the Post-Order traversal sequence of numbers:
10 2 7 12 18 5

(6) The level order traversal of this tree is: 5 7 18 2 12 10

(7) Show the subtree rooted at "7".

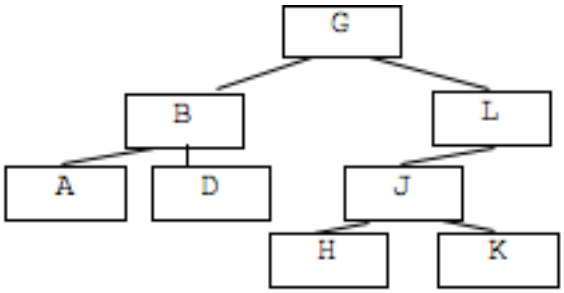


b). Consider the following class defining a Tree Node in a binary tree, and the following mystery method:

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     String val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */

        void mystery(TreeNode n) {
                if (n != null) {
                        System.out.println(n.val);
                        mystery(n.left);
                        System.out.println(n.val);
                        mytery(n.right);
                }
        }
```

Suppose that the above method is called with the root of the following tree as argument. Give the output of this method.



GBAABDDGLJHHJKKL

c). Please finish implementing the following method which finds and returns the number of leaf nodes contained in a binary tree.

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
```

```
 */

public int getLeafNodeCount(TreeNode root){
        // YOUR CODES
if (root==null) return 0;
if (root.left==null && root.right==null) return 1;
return getLeafNodesCount(root.left)+getLeafNodesCount(root.right);



}
```

d). Please finish implementing the following method which determines whether two given binary trees are identical to each other. Two trees are said to be *identical* if the two roots have the same values, and the corresponding left and right children are identical to each other.

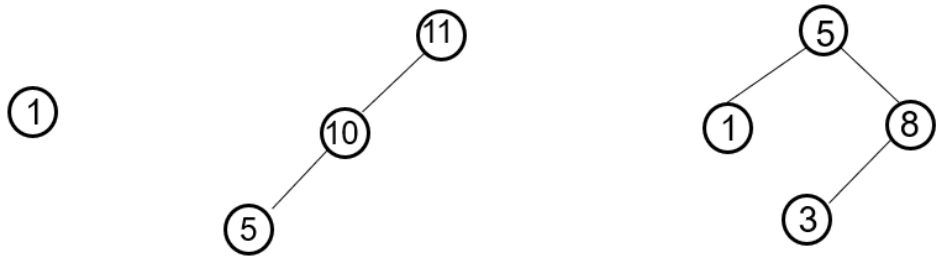
```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */

public boolean isIdentical(TreeNode root1, TreeNode root2){
        // YOUR CODES
if (root1==null && root2==null) return true;
if (root1==null || root2==null) return false;
if (root1.val!=root2.val) return false;
return isIdentical(root1.left, root2.left) && isIdentical(root1.right, root2.right);

}
```

Problem 2 Binary Search Tree

a). For each of the following, determine if it is a binary search tree or not.

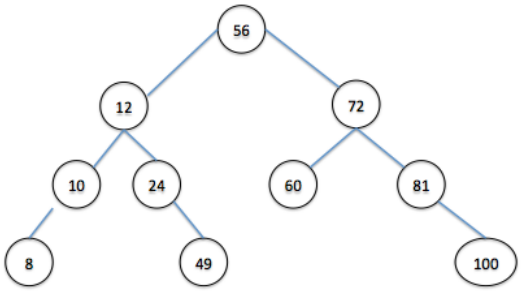


the first and second are. The third is not, because 3 is less than 5.

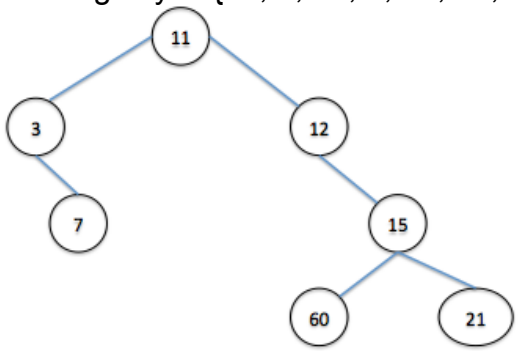
b). Suppose that we have a binary search tree whose keys are integers. A preorder traversal of this tree returns the following keys:

56 12 10 8 24 49 72 60 81 100

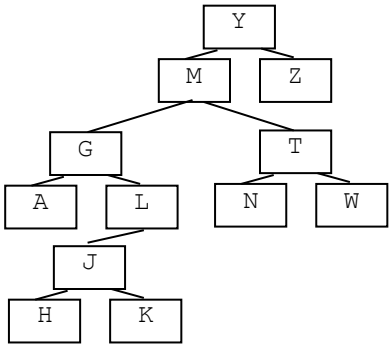
Draw a picture of the tree.



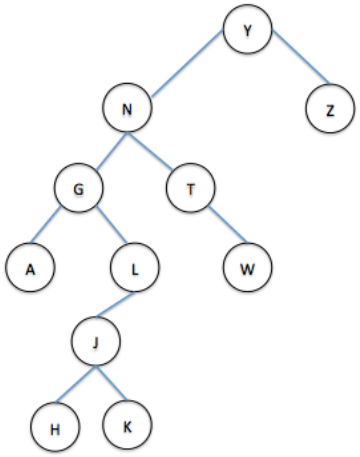
c). Suppose we want to build a binary search tree through successive insertions of the following keys: {11, 3, 12, 7, 15, 21, 13}. Draw the resulting binary search tree.



d). Consider the following binary search tree (note that only the key of each node is shown):



Suppose that node M gets removed. Draw a picture of the resulting binary search tree.



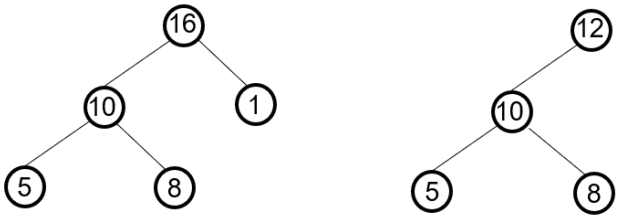
e). Consider the following class defining a tree node in a binary search tree. Finish implementing the following method, so that the minimum value in the binary search tree can be returned.

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public int getMin(TreeNode root){
        // YOUR CODES
if (root==null) return Integer.MIN_VALUE;
while (root.left!=null) root = root.left;
return root.val;
}
```

Problem 3. Binary Heap and Heap Sort
a). Is every binary heap balanced? Justify your answer briefly.

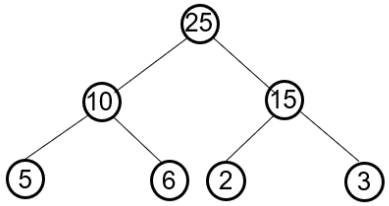
Yes. It is.

b). For each of the following, determine if it is a heap or not.

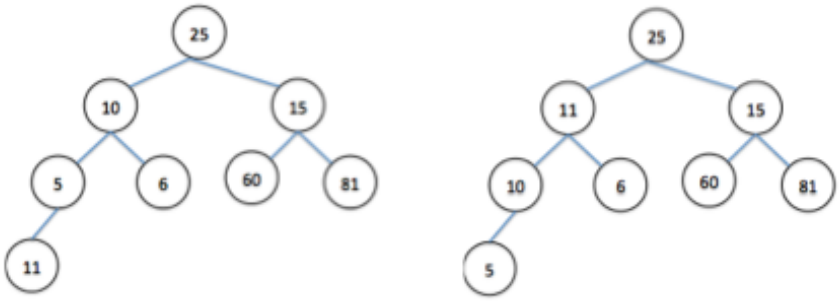


Left one is. (Max-Heap)
Right one is NOT, as it is not a complete binary tree.

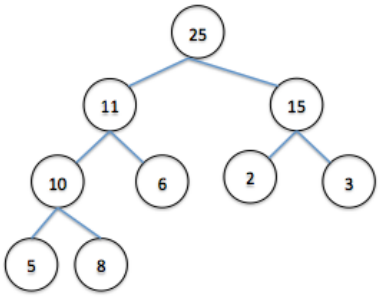
[4pts] Consider the following heap. Draw the heap after these three successive operations:
insert(11)
insert(8)
remove()



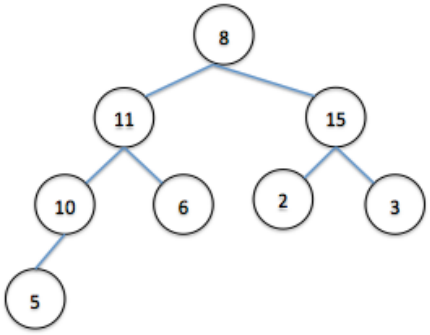
to insert 11, first insert it to be the last node in the tree, and then possibly trickle it up.



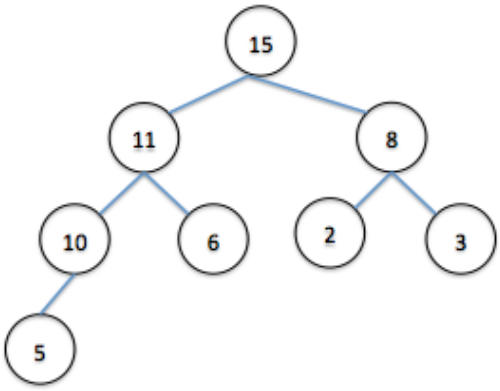
to insert 8, follow the same process. First add it to be the last node on the tree, and then possibly trickle it up. Here in this example, we do not need to trickle it up.



To remove, we always remove the root node. We will exchange it with the last node on the tree, and remove the original root node. The resulting tree would be:



and then trickle down the node 8, resulting in the following tree:



c). Given the following array representation of a binary heap:

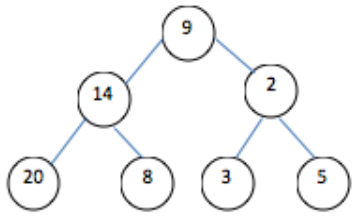
30	22	25	14	5	20	7	3	9	1	2	6

What is the left child of 20?____6____

What is the right child of 14?___9____

What is the parent of 2?__5_____

d). Given the following complete binary tree, draw the **max-heap** that can be constructed from it.



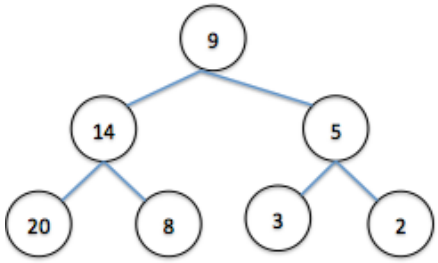
There are 7 nodes on this tree. So we start at nodes with index (7-1)/2 = 3, which is the node 20 to start the reheapification process.

The corresponding codes is:

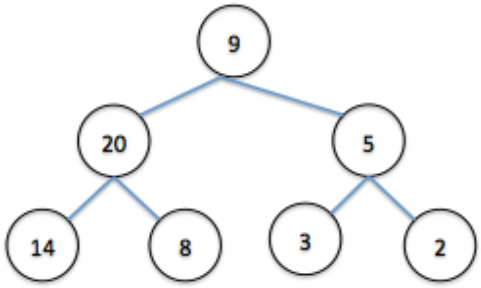
```
for (int i= (n-1)/2; i>=0; i--){
        trickle_down(i)
}
```

1) we will first process on node 20. There is no need to trickle it down.

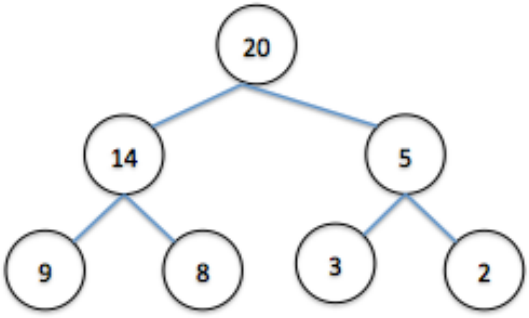
2) We then check on node 2, and trickle it down.



3) we then check on node 14, and trickle it down.



4) we then check on node 9, and trickle it down.

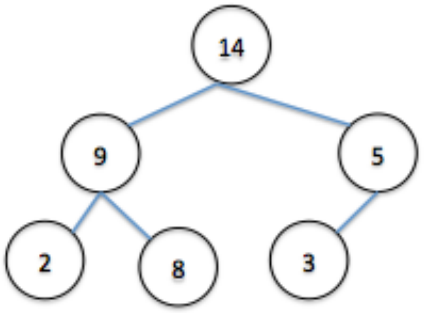


e). Suppose the elements of an array are in sorted order. Must it be a heap? Yes

f) Suppose we are sorting an array {9, 14, 2, 20, 8, 3, 5} using heap sort. Show the content of the array for the first two iterations.
After heapifying the complete binary tree into a max-heap as what we did in step d), we can first exchange the root node (20) with the last node on the tree (2), so the array after 1st

iteration would be:
[2, 14, 5, 9, 8, 3, 20]

Since 2 is now at the root, we need to trickle it down to reheapify the tree. The resulting tree will look like the following:



exchange the root node (14) with the last node (3). The array after 2nd iteration would be:
[3, 9, 5, 2, 8, 14, 20]

Problem 4. Hash Table

a). Consider linear probing, quadratic probing, and double hashing methods for collision resolution. Show the content of the hash table after inserting the keys listed below. Assume the hash functions is h(k) = k % 7. For the double hashing method, the second hash function (to determine the step size) is $h_2(k)$= (k % 5) + 1

For using open addressing for collision solving, the general formula to find the next vacant cell is always be:

h = (h + s) % tableSize

where:

1) in Linear Probing, s =1
2) in Quadratic Probing:
   s= 1^2 = 1 in the first iteration
   s = 2^2 in the second iteration
   s = 3^2 in the third iteration
   …..
3) in Double hashing, the s is determined by the second hash function.

Insert (from left to right): **11, 4, 19, 20, 29**

Linear Probing:

0	1	2	3	4	5	6
20	29			11	4	19

Quadratic Probing:

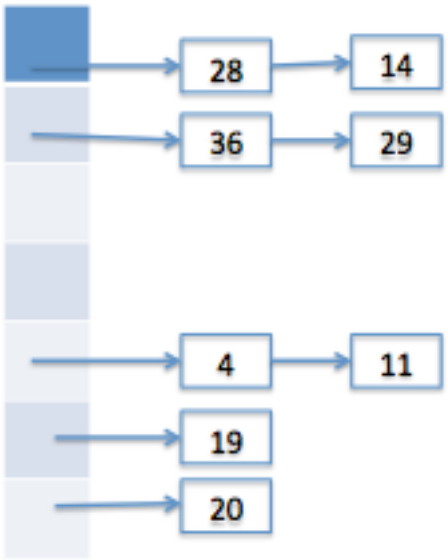
0	1	2	3	4	5	6
20	29			11	4	19

Double Hashing:

0	1	2	3	4	5	6
	29	4		11	19	20

b). Consider separate chaining methods for collision resolution. Show the content of the hash table after inserting the keys listed below. Assume the hash functions is
h(k) = k % 7.
Insert (from left to right): 11, 4, 19, 20, 29, 14, 28, 36

Suppose we maintain the linked list by always inserting at the beginning. The separate chain would be:



c). Consider the following hash table, which uses linear probing to resolve collisions.

19	20		13	23	55	26	14	73	64

Give a sequence of insertions that would have caused the hash table to look like that.

One possible answer is: 13 23 55 26 14 73 64 19 20

d). Try to write a HashTable application client (can either use the Java built-in Hashtable or HashMap class) to solve the following problem:

Given an array of integers, find if the array contains any duplicates. Your function should return true if any value appears at least twice in the array, and it should return false if every element is distinct.

```
public boolean containDuplicate(int[] arr){
        HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
        for (int num: arr){
                if (!map.containsKey(num)){
                        map.put(num, 1);
                } else { return false;}
        }
        return true;
}
```